

D21

Semestral task

peer review

Audit

5th December 2021

by Blockchain student Bc. Jan Smrža from UTB




Table of Contents

1. Overview	2
1.1 Ackee Blockchain	2
1.2 Audit Methodology	2
1.3 Review team	3
1.4 Disclaimer	3
2. Scope	4
2.1 Coverage	4
2.2 Supporting Documentation	4
2.3 Objectives	4
3. System Overview	5
D21.sol	5
IVoteD21.sol	6
4. Security Specification	7
4.1 Actors	7
4.2 Trust model	7
5. Findings	8
5.1 General Comments	8
5.2 Issues	8
5.3 Unit testing	13
5.4 Static analysis	14
6. Conclusion	15
Appendix A	16

Document Revisions

Version	Date	Description
1	2021/12/3	Research, pre-audit version
2	2021/12/4	Critical and other issues report, unit testing
3	2021/12/5	Static analysis, final report

1. Overview

This document presents my findings in reviewed contracts.

1.1 Ackee Blockchain

[Ackee Blockchain](#) is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course [Summer School of Solidity](#) and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, [Rockaway Blockchain Fund](#).

1.2 Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools Mythril and Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment analysis** - the contracts are deployed locally in various versions to determine the most efficient gas usage.
5. **Unit testing** - additional unit tests are written in the Brownie testing framework to ensure the system works as expected.

1.3 Review team

The audit has been performed with a total time donation of 3 engineering days. The whole process was supervised by the Audit Supervisor.

Member's Name	Position
Bc. Jan Smrža	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

1.4 Disclaimer

Best effort has been put to find all vulnerabilities in the system, however the findings shouldn't be considered as a complete list of all existing issues.

2. Scope

This chapter describes the audit scope, contains provided specification, used documentation and set main objectives for the audit process.

2.1 Coverage

Files being audited:

- D21.sol
- IVoteD21.sol

Sources revision used during the whole auditing process:

- Repository: **Private**
- Commit: **Basic solution, 18th November 2021**
- Audited contract included in audit repository

2.2 Supporting Documentation

Developer provided a tutorial explaining the required code structure and the description of the given problem. They also provided a more detailed description at <https://www.ih21.org/o-metode>

Source code comments were also used for my understanding of the system and implementation if it fits the given task.

2.3 Objectives

The following main objectives of the audit have been defined:

- Check the code quality, architecture and best practices.
- Check the gas efficiency of the contract
- Check if contract is working as intended according to given set of rules
- Check the security of the contract
- Check the contract with unit testing using brownie
- Check the contract with static analysis

3. System Overview

This chapter describes the audited system. Contract is split in an interface defining functions and structures and a base contract implementing this abstract class. The voting system is supposed to work as follows:

- voting starts with the deployment of the system and lasts for one week
- only owner (chairperson) can add eligible voters
- each voter has two votes, 2 positive and 1 negative
- they vote for subjects which can be added by any person or party
- anyone can request the subject name and vote count during any time of the elections
- positive votes cannot be given to the same subject
- negative vote can only be given after the person assigns both positive votes
- it is not specified whether a negative vote can be given to subject a person already voted positive for

D21.sol

Base contract for the `voting system`. Contains contract constructor, structures, mappings, functions, addresses. Implements the ability to add a subject, add a voter, vote for subjects, list subjects and get a remaining time until elections are finished.

The functions are implemented as follows:

- function `addSubject(string memory _name)` external
 - subject can be added by any address. If this address adds a subject they cannot add more
- function `addVoter(address addr)` external
 - only owner can add a voter
- function `getSubjects()` external view returns `(address[] memory)`
 - anyone can call a function to get addresses of all listed subjects, however the call is reverted if there are no subjects listed yet
- function `getSubject(address addr)` external view returns `(Subject memory)`
 - anyone can call a function to get one subject's name and vote count, however if the subject address is not found, the call is reverted
- function `votePositive(address addr)` external
 - this function implements all the requirements of giving a positive vote to a subject and removing this positive vote from a voter
- function `voteNegative(address addr)` external
 - ensure a person can vote negative after distributing positive votes

- function `getRemainingTime()` public view returns (uint256)
 - this function returns the remaining time until the elections are done

Immutable owner and ending time are assigned in the constructor during contract deployment. Developer is using mappings to assign addresses to voter structure and addresses to subject structure. All subject addresses are stored in an array for listing.

IVoteD21.sol

Abstract contract which is used as a parent of `D21`. Contains only functions' definitions with no implementation and two structures:

- struct Voter
- struct Subject

4.Security Specification

This section specifies single roles and their relationships in terms of security in my understanding of the audited system. These understandings are later validated.

4.1 Actors

This part describes actors of the system, their roles and permissions.

Owner

Immutable owner deploys the contracts into the network. To my understanding it is someone to be trusted, a chairperson or representative of an office. They cannot be changed later and they are the only ones who can add eligible voters. Thus this is an important role and the owner must be chosen wisely.

Voter

Every assigned voter has two positive and one negative vote. They are the only ones who can vote.

Subject creator

Subject creators are the people who list a subject, thus they are the representatives of the party and possibly the election winners (or losers).

Public

Anyone from the public has access to see the current subjects, their names and vote counts. Also they can ask how much time remains until the elections are complete.

4.2 Trust model

Everyone has to trust the contract owner in terms of distributing the right to vote accordingly. This right to vote is a critical security issue found described further below.

5. Findings

This chapter shows detailed output of my analysis and testing.

5.1 General Comments

The code does have general comments and everything seems understandable. Although it could be described in more detail. Code quality follows basic good practices and altogether is above average. Nevertheless, there are parts which are missing necessary requirements, parts which result in error and parts which are written illogically.

5.2 Issues

Using my toolset, manual code review and unit testing the following issues have been identified. They are split in Informational, Low, Medium, High and Critical categories.

Informational

Informational issues only comments and aesthetics. I provide hints on how to improve code readability and follow best practices. Further actions depend on the development team decision.

In general, most of the reported informational issues are concerning commenting on the code itself. An auditor must seek outside sources to understand the voting system precisely. More clarity would be welcomed.

ID	Description	Contract	Line	Status
I1	Structure comments missing in declaration	IVoteD21.sol	5, 11	Reported
I2	More detailed usage comments in declaration, NatSpec Format	IVoteD21.sol	-	Reported
I3	voteNegative(address addr) : text issue	D21.sol	66	Reported
I4	missing a function to return the winning subjects	D21.sol, IVoteD21.sol	-	Reported

I1: In interface IVoteD21 the defined structures are missing comments completely.

I2: I recommend expanding the comment base in declaration, describing into more detail what each function is supposed to do, its limitations and requirements. NatSpec Format comments would be very welcomed.

I3: A voter who forgot if they already voted negative can be confused by the returning string of the following line:

```
require(voters[msg.sender].votes == 2, "For the permission to vote negatively, U have to vote positively twice, first");
```

which blocks the voter from voting negative sooner than voting positive twice or blocks the voter from voting negative twice. They might try to vote positive again. Recommendation: either split the require in two, or rewrite the return string to be more ambiguous.

I4: Although it has not been given in the task for the developer, it might have been worth it to add a custom function to return the ranking of a few winning subjects (top 5 for example). This way, if someone wants to check who is winning, they have to call the function `getSubject()` as many times as there are registered subjects. This might be very ineffective.

Low

Low severity issues are more comments and recommendations rather than security issues. They include low gas leaks and illogical, but in no way dangerous parts of the code.

ID	Description	Contract	Line	Status
L1	<code>getSubject(address addr) : gas leak</code>	D21.sol	43-46	Reported
L2	<code>addSubject(string memory _name) : empty string</code>	D21.sol	20-28	Reported
L3	<code>votePositive(address addr) : gas leak</code>	D21.sol	57, 58	Reported
L4	<code>voteNegative(address addr) : gas leak</code>	D21.sol	68, 69	Reported
L5	<code>votePositive(address addr) : gas leak</code>	D21.sol	49-59	Reported
L6	<code>voteNegative(address addr) : gas leak</code>	D21.sol	62-70	Reported

L7	addSubject(string memory _name): 'if' statement not necessary	D21.sol	22-24	Reported
----	---	---------	-------	----------

L1: It is not necessary to ensure a subject address is created when calling a `getSubject()` function. By simply calling the function with an unregistered subject, it returns an empty tuple. Overall, the function will be more effective and gas usage reduced. I recommend following steps. Remove line 44 completely:

```
require(keccak256(bytes(subjects[addr].name)) != (keccak256(bytes(""))),
"There is no registered subject on the given address");
```

L2: In function `addSubject()` it is possible to add a subject with no name (empty string). I recommend asking for the subject name to be at least 2 characters. Eg.:

```
require(bytes(_name).length > 1, "...");
```

L3: In function `votePositive()` the vote counts for voter and subject are incremented as follows: `voters[msg.sender].votes += 1;` and `subjects[addr].votes++;` I recommend pre-increment operator which is the most gas efficient way.

L4: In function `voteNegative()` the vote counts for voter and subject are incremented as follows: `voters[msg.sender].votes++;` and `subjects[addr].votes--;` I recommend pre-increment operator which is the most gas efficient way.

L5 and L6: In functions `votePositive()` and `voteNegative()` voter structure is accessed with index multiple times. It is more efficient to use storage and use the loaded voter variable instead. Eg.: `Voter storage vot = voters[msg.sender];`

L7: In functions `addSubject()` 'if' statement is used before a require statement. This can be rewritten removing the 'if' statement this way:

```
require(keccak256(bytes(subjects[msg.sender].name)) ==
(keccak256(bytes("")))) , "U have already registered a subject");
```

However, it does not improve any gas usage so this is just an aesthetical thing.

Medium

Medium severity issues aren't security vulnerabilities, but should be clearly clarified or fixed. These issues might also include the effectiveness of the contract considering gas usage.

ID	Description	Contract	Line	Status
M1	using keccak256 to verify addresses	D21.sol	22, 44, 51, 64	Reported
M2	possible to add voters and subjects after elections have ended	D21.sol	20-28, 31-34	Reported
M3	calling getRemainingTime() function from voteNegative(address addr) and votePositive(address addr)	D21.sol	50, 63	Reported

M1: Functions `addSubject()`, `getSubject()`, `votePositive()`, `voteNegative()` are all using `keccak256` to verify the fact that a subject is registered. It is possible to remove `keccak256` completely by mapping subject addresses to a boolean. Although it is true, setting the value of this mapping from false to true when creating a subject costs more gas than using `keccak256` inplace, but later, while using this mapping during voting positive or negative, it is more cost efficient. So it must be evaluated if the system is going to have much more voters than subject creators. Eg.: if there will be 10 subjects and 5 voters (inefficient to have mapping), or if there will be 10 subjects and 1000 voters (more efficient to have mapping). Thus I recommend considering this option and possibly replacing `keccak256` with mapping of subject addresses to boolean.

M2: In functions `addSubject()`, `addVoter()` it is possible to add subjects and voters after the elections have ended. Although a voter cannot vote after the elections have ended, the issue is greater when a new subject is created. Consider subjects which have a negative amount of votes. The newly created subject will have 0 votes and suddenly climb the ranking ladder above all these downvoted subjects. Thus I recommend adding a requirement or modifier to these functions to eliminate usage after elections are done.

M3: In functions `votePositive()`, `voteNegative()` a function `getRemainingTime()` is used. It is much more efficient to rather use the same condition as in `getRemainingTime()` than calling the function itself returning a value.

Eg. replace:

```
require(getRemainingTime() > 0, "Voting time has finished!");
```

with

```
require(int(endTime) - int(block.timestamp) > 0, "Voting time has finished!");
```

High

High severity issues are vulnerabilities, which require specific steps and conditions to be exploited. They include errors. These issues have to be fixed.

ID	Description	Contract	Line	Status
H1	getRemainingTime(): error	D21.sol	74	Reported
H2	voteNegative(address addr): error	D21.sol	63	Reported
H3	votePositive(address addr): error	D21.sol	50	Reported

H1: `getRemainingTime()` works with an unsigned integer variable. By default `block.timestamp` is also an unsigned value. On line 74 in this function:

```
require((endTime - block.timestamp) > 0, "Voting time has finished!");
```

an immutable unsigned variable which is assigned in contract constructor is diminished by a `block.timestamp` to evaluate if the elections have finished. As soon as seven days pass, this function's `require` starts failing ever since, because the resulting value is negative, thus reverts completely.

In brownie, it throws a `brownie.exceptions.VirtualMachineError` exception which can be caught by using `brownie revert`: `brownie.reverts("Integer overflow")`.

I recommend checking if `endTime` is greater than `block.timestamp` in order for this `require` to pass:

```
require(endTime > block.timestamp, "Voting time has finished!");
```

H2 and H3: Since functions `voteNegative(address addr)` and `votePositive(address addr)` are utilizing function `getRemainingTime()`, they are bound to fail as well after the elections have ended.

Critical

Direct critical security threats, which could be instantly misused to abuse the system. These issues must be fixed.

ID	Description	Contract	Line	Status
C1	addVoter(address addr) : abuse	D21.sol	31-34	Reported

C1: The main goal of the contract in whole is supposed to ensure an effective, unbiased and secure way to elect representatives. There is a huge responsibility on the contract owner who is the only person eligible to add new voters who are responsible for the most important part of the contract. The owner can successfully add a new voter. However, once the voter is added (and they use all of their votes), the way the contract is built the owner can add the same voter (address) once again to reset their vote counts, thus allowing the same person to vote over and over again. I consider this as a critical security issue, because if the contract owner (chairperson) is not to be trusted or bribed, they can easily distribute more voting rights to certain groups of people and this way, destroying the whole sense of D21 voting method. I recommend adding this voter to a mapping to evaluate if they had already been added before trying to re-add them later. Unit test U6.

5.3 Unit testing

There are a lot of test cases built to cover all of the code if possible (viz [Appendix A](#)), however, due to the fact that developer is not using public variables (owner, endTime, voters, subjects, subjectsAddresses) I could not assert the correctness of these as I could not access them without rewriting the contract itself.

Overview

In total 31 unit tests were run covering all possible branches of each function. 3 of these tests are bound to fail intentionally, because they simulate the problem with unsigned variables as mentioned in [High severity issues](#). This is also demonstrated in [Appendix A](#) using the brownie console.

```
test_addSubject_anyone PASSED
test_addSubject_same PASSED
test_addSubject_expired PASSED
test_addVoter_owner PASSED
test_addVoter_nonOwner PASSED
test_addVoter_readd PASSED
test_addVoter_expired PASSED
test_getSubjects_empty PASSED
test_getSubjects_registered PASSED
test_getSubject_empty PASSED
test_getSubject_registered PASSED
test_votePositive_expired PASSED
test_votePositive_expired_v2 PASSED
test_votePositive_nonRegistered PASSED
test_votePositive_noRightToVote PASSED
test_votePositive_success PASSED
test_votePositive_sameSubject PASSED
test_votePositive_moreThanTwice PASSED
test_voteNegative_expired PASSED
test_voteNegative_expired_v2 PASSED
test_voteNegative_nonRegistered PASSED
test_voteNegative_noRightToVote PASSED
test_voteNegative_noTwoPositiveVotes PASSED
test_voteNegative_success PASSED
test_voteNegative_twice PASSED
test_voteNegative_sameSubject PASSED
test_getRemainingTime_success PASSED
test_getRemainingTime_fail PASSED
test_getRemainingTime_boundToFail FAILED
test_voteNegative_fail FAILED
test_votePositive_fail FAILED
```

5.4 Static analysis

Slither and Mythril both recognized the usage of `block.timestamp`, a predictable variable and warned against using it in comparisons and control flow. However, this functionality cannot be achieved any different way. This is not an issue.

```
Using slither version: 0.8.1
Refreshing explorer...
Loaded 0 issues, displaying 0
—— Starting analysis ——
✗ D21.votePositive(address) (D21.sol:50-60) uses timestamp for comparisons
  • require(bool,string)(getRemainingTime() > 0,Voting time has finished!) (D21.sol#51)

✗ D21.voteNegative(address) (D21.sol:63-71) uses timestamp for comparisons
  • require(bool,string)(getRemainingTime() > 0,Voting time has finished!) (D21.sol#64)

✗ D21.getRemainingTime() (D21.sol:74-77) uses timestamp for comparisons
  • require(bool,string)((endTime - block.timestamp) > 0,Voting time has finished!) (D21.sol#75)

✗ Pragma version^0.8.9 (D21.sol:2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6

✗ Pragma version^0.8.9 (IVoted21.sol:2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6

✗ solc-0.8.10 is not recommended for deployment

✗ Parameter D21.addSubject(string)._name (D21.sol:21) is not in mixedCase
```

```
smrzs@smrzs-IdeaPad-5-14ALC05:~/Blockchain/solcoverage$ myth analyze 021.sol
==== Dependence on predictable environment variable ====
SWC ID: 116
Severity: Low
Contract: 021
Function name: constructor
PC address: 228
Estimated Gas Usage: 230 - 420
A control flow decision is made based on The block.timestamp environment variable.
The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable
and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be a
ware that use of these variables introduces a certain level of trust into miners.
.....
```

6. Conclusion

I started with reviewing the code manually with local smart contract deployment and running manual tests to determine things which might be broken. Afterwards I wrote brownie unit tests to cover as many scenarios as possible for this type of a contract. At last, static analysis was done using Slither and Mythril. In total, 5 Informational, 7 Low, 3 Medium, 3 High and 1 Critical severity issues were found. I recommend fixing the Critical and High issues as soon as possible. The developer was informed of these issues.

Appendix A

Demonstration of contract's behaviour leading to a failed U29, U30 and U31 tests.

```
Brownie v1.17.1 - Python development framework for Ethereum

AuditSmrzaProject is the active project.

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul --mnemonic brownie'...
Brownie environment is ready.
>>> cont = D21.deploy({'from': accounts[1]})
Transaction sent: 0xd8455e7204d5583c93d1937bf773bbb5d2aab110fb31f334bf1350aa9175bd1c
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
D21.constructor confirmed Block: 1 Gas used: 867092 (7.23%)
D21 deployed at: 0xe7CB1c67752cBb975a56815Af242ce2Ce63d3113

>>> cont.getRemainingTime
<ContractCall 'getRemainingTime()'>
>>> cont.getRemainingTime()
604800
>>> chain.sleep(604800)
>>> chain.mine()
2
>>> cont.getRemainingTime()
File "<console>", line 1, in <module>
File "brownie/network/multicall.py", line 115, in _proxy_call
    result = ContractCall._call(*args, **kwargs) # type: ignore
File "brownie/network/contract.py", line 1661, in _call
    return self.call(*args, block_identifier=block_identifier)
File "brownie/network/contract.py", line 1457, in call
    raise VirtualMachineError(e) from None
VirtualMachineError: revert
>>> accounts
[<Account '0x66aB6D9362d4F35596279692F0251Db635165871'>, <Account '0x33A4622B82D4c04a53e170c638B944ce27cffce3'>, <Account '0x21b42413bA931038f35e7A5224FaDb065d297Ba3'>, <Account '0x46C0a5326E643E4f71D3149d50B48216e174Ae84'>, <Account '0x86426F076647A5362706a04570A5965473B'>, <Account '0x23BB2Bb6c340D4C91cAa478EdF6593fC5c4a6d4B'>, <Account '0xA868bC7c1Be5bf2507a92a755dcF1D8e8dc'>]
>>> cont.addVoter(accounts[2], {'from': accounts[1]})
Transaction sent: 0x6db92b59cab794a1537ee6960a72cdadcc441c2e6da2132516c1a08a493ebd30
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 1
D21.addVoter confirmed Block: 3 Gas used: 42942 (0.36%)

<Transaction '0x6db92b59cab794a1537ee6960a72cdadcc441c2e6da2132516c1a08a493ebd30'>
>>> cont.addSubject("test", {'from': accounts[1]})
Transaction sent: 0x2cefee216133c8e1ec561d78687d56fa149cf35f0e0f28b1aa01cd774a8675db
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 2
D21.addSubject confirmed Block: 4 Gas used: 87156 (0.73%)

<Transaction '0x2cefee216133c8e1ec561d78687d56fa149cf35f0e0f28b1aa01cd774a8675db'>
>>> cont.votePositive(accounts[1], {'from': accounts[2]})
Transaction sent: 0x18968728de2cbfffb38f03a72a4951c8475ff895efe7011276cb13a3037dfe937
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
D21.votePositive confirmed (Integer overflow) Block: 5 Gas used: 21830 (0.18%)

<Transaction '0x18968728de2cbfffb38f03a72a4951c8475ff895efe7011276cb13a3037dfe937'>
>>> 
```

Coverage results

Since brownie has an issue with immutable variables and also does not recognize coverage of one function, I was forced to remove immutable modifier from owner and endTime variables. The function not recognized for coverage by brownie is getSubjects() which is covered by tests U8 and U9. Also, it is not possible to hit 100% coverage, because of the errors demonstrated in console in a figure above and by tests U29, U30 and U31.

```

contract: D21 - 94.2%
  D21.addSubject - 100.0%
  D21.addVoter - 100.0%
  D21.getSubject - 100.0%
  D21.votePositive - 95.0%
  D21.voteNegative - 93.8%
  D21.getRemainingTime - 75.0%

```

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.9;
3 import "./IVoteD21.sol";
4
5 contract D21 is IVoteD21 {
6     address owner;
7     uint256 endTime;
8
9     mapping(address => Voter) voters;
10    mapping(address => Subject) subjects;
11    address[] subjectsAddresses;
12
13
14    constructor() {
15        owner = msg.sender;
16        endTime = block.timestamp + 1 weeks;
17    }
18
19    // Add a new subject into the voting system using the name.
20    function addSubject(string memory _name) external {
21        // for some reason does not work without if condition
22        if(keccak256(bytes(subjects[msg.sender].name)) != (keccak256(bytes("")))) {
23            require(false, "U have already registered a subject");
24        }
25
26        subjects[msg.sender] = Subject(_name, 0);
27        subjectsAddresses.push(msg.sender);
28    }
29
30    // Add a new voter into the voting system
31    function addVoter(address addr) external {
32        require(msg.sender == owner, "Only owner can give right to vote.");
33        voters[addr] = Voter(true, address(0x0), 0);
34    }
35
36    // Get addresses of all registered subjects
37    function getSubjects() external view returns (address[] memory) {
38        require(subjectsAddresses.length > 0, "There is no subject yet");
39        return subjectsAddresses;
40    }
41
42    // Get the subject details.
43    function getSubject(address addr) external view returns (Subject memory) {
44        require(keccak256(bytes(subjects[addr].name)) != (keccak256(bytes("")))), "There
45        return subjects[addr];
46    }
47
48    // Vote positive for the subject.
49    function votePositive(address addr) external {
50        require(getRemainingTime() > 0, "Voting time has finished!");
51        require(keccak256(bytes(subjects[addr].name)) != (keccak256(bytes("")))), "There
52        require(voters[msg.sender].rightToVote, "U dont have right to vote");
53        require(voters[msg.sender].firstPositive != addr, "U already voted positively fo
54        require(voters[msg.sender].votes < 2, "U already voted positively twice");
55
56        voters[msg.sender].firstPositive = addr;
57        voters[msg.sender].votes += 1;
58        subjects[addr].votes++;
59    }
60
61    // Vote negative for the subject.
62    function voteNegative(address addr) external {
63        require(getRemainingTime() > 0, "Voting time has finished!");
64        require(keccak256(bytes(subjects[addr].name)) != (keccak256(bytes("")))), "There
65        require(voters[msg.sender].rightToVote, "U dont have right to vote");
66        require(voters[msg.sender].votes == 2, "For the permission to vote negatively, U
67
68        voters[msg.sender].votes++;
69        subjects[addr].votes--;
70    }
71
72    // Get remaining time to the voting end in seconds.
73    function getRemainingTime() public view returns (uint256) {
74        require((endTime - block.timestamp) > 0, "Voting time has finished!");
75        return (endTime - block.timestamp);
76    }
77}

```

Thank You

Ackee Blockchain a.s.



Prague, Czech Republic



hello@ackeeblockchain.com



<https://discord.gg/z4KDUbuPxq>